

# University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Summer 2015

Instructor: Sagar Karandikar

2015-07-09



# CS61C MIDTERM 1



After the exam, indicate on the line above where you fall in the emotion spectrum between “sad” & “smiley”...

<i>Last Name</i>	
<i>First Name</i>	
<i>Student ID Number</i>	
<i>Login</i>	<b>cs61c-</b>
<i>The name of your <b>SECTION</b> TA (please circle)</i>	Derek   Harrison   Jeffrey   Nathaniel   Rebecca
<i>Name of the person to your Left</i>	
<i>Name of the person to your Right</i>	
<i>All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61C who have not taken it yet. (please sign)</i>	

## Instructions (Read Me!)

- This exam contains 8 numbered pages including the cover page. **The back of each page is blank and can be used for scratch-work but will not be looked at for grading.** (i.e. the sides of pages without the printed “SID: \_\_\_\_\_” header will not even be scanned into Gradescope).
- Please turn off all cell phones, smartwatches, and other mobile devices. Remove all hats & headphones. Place your backpacks, laptops and jackets under your seat.
- You have 80 minutes to complete this exam. The exam is closed book; you may not use any computers, phones, wearable devices, or calculators. You may use one page (US Letter, front and back) of handwritten notes in addition to the provided green sheet.
- There may be partial credit for incomplete answers; write as much of the solution as you can. We will deduct points if your solution is far more complicated than necessary. When we provide a blank, please fit your answer within the space provided. “IEC format” refers to the mebi, tebi, etc prefixes.

	<b>Q1</b>	<b>Q2</b>	<b>Q3</b>	<b>Q4</b>	<b>Q5</b>	<b>Q6</b>	<b>Total</b>
Points Possible	13	13	14	15	20	15	90

**Q1) Number Representation (13 pts)**

a) Fill in the blanks with a letter (a, b, c or d) to match each expression in the left column, with an equivalent expression from the right column:

- |      |                       |   |       |                             |
|------|-----------------------|---|-------|-----------------------------|
| i)   | $(x \gg 16) \ll 16$   | = | _____ | a. 0                        |
| ii)  | $x \wedge x \wedge x$ | = | _____ | b. $\sim x$                 |
| iii) | $x \wedge -1$         | = | _____ | c. $x$                      |
| iv)  | $x \& \sim x$         | = | _____ | d. $x \& 0x\text{FFFF}0000$ |

b) Rewrite the following numbers using IEC prefixes or as approximations using IEC prefixes:

$$1024 = \underline{\hspace{2cm}} \quad 10^6 = \underline{\hspace{2cm}} \quad 2^{43} = \underline{\hspace{2cm}}$$

c) Convert the following 8 bit two's complement numbers into their decimal equivalents.

$$0x80 = \underline{\hspace{2cm}} \quad 0x7E = \underline{\hspace{2cm}}$$

d) Four's complement is very similar to two's complement in the sense that negation consists of flipping digits and adding one to the "flipped" version of the value. To flip a value in 4's complement, 0's become 3's, 1's become 2's, and vice versa.

$$\text{e.g. } 3332_4 = -(-3332_4) = -(0001_4 + 1) = -(0002_4) = -(0 + 0 + 0 + 2 \cdot 4^0) = -2$$

A quaternary (base 4) digit is known as a crumb (in relation to bytes, nibbles, and bits). For the following questions, we would like to convert the following **4-crumb 4's complement** numbers (0cXXXX) to decimal. You may leave values in expression format. (Hint: How do we convert from hex to binary?)

$$0c2000 = \underline{\hspace{2cm}} \quad 0c1333 = \underline{\hspace{2cm}}$$

Now using **8-crumb 4's complement**, what would the quaternary representation of the following be?

$$270 = \underline{\hspace{2cm}} \quad -17 = \underline{\hspace{2cm}}$$

**Q2) Pointers and Memory Management (13 pts)**

In this question, assume mallocs are always successful, pointers are 4 bytes, ints are 4 bytes, doubles are 8 bytes, and chars are 1 byte.

a) Which is the most efficient implementation of f() given below? Explain in a sentence or two.

```
typedef struct { int vals[1000000]; } dataStruct;
```

```
1. int f(dataStruct d){ printf("%d", d.vals[0]); }
2. int f(dataStruct *d){ printf("%d", d->vals[0]); }
3. int f(dataStruct **d){ printf("%d", (*d)->vals[0]); }
```

# \_\_\_\_\_ Explain: \_\_\_\_\_

b) Some of the code samples below contain issues with memory management. Below, identify the first instance of such a mistake in each code sample and briefly describe the issue. If there is no issue in the code sample, write "None".

```
i.) double *pi_ptr;
pi_ptr = malloc(sizeof(pi_ptr));
*pi_ptr = 3.14;
```

Circle above and explain here: \_\_\_\_\_

```
ii.) char *a = "abcdef";
char **c = &a;
int x = 0;
while(**c != NULL){
    printf("%s", *(c + x));
    x++;
}
```

Circle above and explain here: \_\_\_\_\_

c) Below are two renditions of similar code for a simple function `foo()`. Identify the number of bytes stored in the stack, heap, and static up to the line marked with “← here” for each piece of code. Include allocations only from the lines shown; assume registers are not used. You can assume that all calls to `malloc` succeed (do not return `NULL`).

**i) Snippet #1:**

```
char * str = "this string";

int foo (int a) {
    char* thisPtr = malloc(sizeof(char) * 12);
    thisPtr = str;
    char* thatPtr = "that string";
    int i = 0;
    while (thisPtr[i]) {
        if (thisPtr[i] != thatPtr[i])
            thisPtr[i] = thatPtr[i];
        i++;
    }
    ...// ← here
```

Stack: \_\_\_\_\_

Heap: \_\_\_\_\_

Static: \_\_\_\_\_

**ii) Snippet #2:** Now, follow the same steps as in the previous code snippet for the code below. However, the code below may contain bugs. If you find any potential bugs in the code below, explain what the bug is and then give the number of bytes in stack, heap, and static right before the bug occurs. You may assume that everything `malloc`'d is eventually freed (for example in the “...” section of the code). Therefore, “memory leak” is not a potential bug.

```
char *str = "this string";
char *str2 = "that string";
typedef struct{ char* str; int size;} bigStr;

int foo () {
    bigStr* thisPtr = malloc(sizeof(bigStr));
    bigStr* thatPtr = malloc(sizeof(bigStr));
    thisPtr->str = str;
    thatPtr->str = str2;
    int i;
    for(i=12; i>=0; i--) {
        thatPtr->str[i] = thisPtr->str[i];
    }
    ...// ← here
}
```

Stack: \_\_\_\_\_

Heap: \_\_\_\_\_

Static: \_\_\_\_\_

Errors (if any):

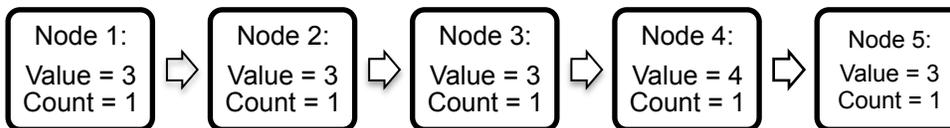
**Q3) Linked List (14 pts)****Run Length Encoding**

Complete the `compress_ll()` function so that it will compress consecutive values in a linked list according to the provided struct below and update the count field in the struct, which represents the number of consecutive copies of the associated value we have seen so far. This compression is known as run-length encoding and is typically used for data compression.

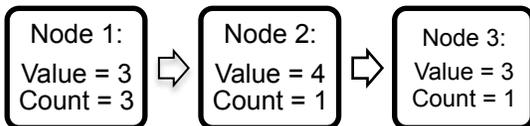
```
struct ll {
    int value;
    int count; // count used for following problem
    struct ll* next; // pointer to next element
};
```

For example, given the lists below:

**Before:**



**After:**



You may assume that we will pass in a valid node so you do not need to check that the initial node passed in is not NULL. You might not need all of the blank lines.

**Note that all list nodes were created via dynamic memory allocation.**

```
void compress_ll(struct ll** node, int curr_value) {
    struct ll* next_node = (*node)->next;

    if ( _____ ) {
        int next_value = next_node->value;

        _____

        if ( _____ ) {
            (*node)->count += 1;

            _____

            _____

            _____
        } else {

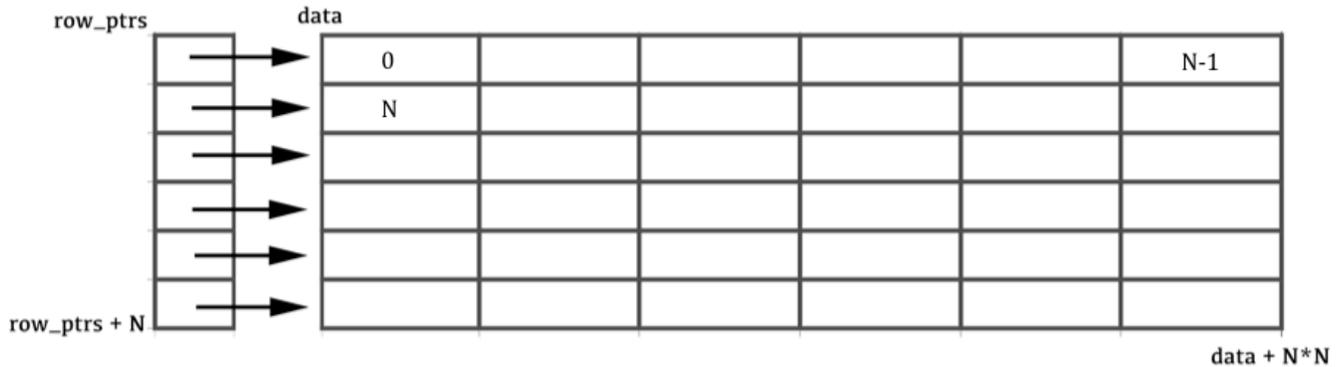
            _____

            _____

            _____
        }
    }
}
```

**Q4) Array Hacking (15 pts)**

Write code to dynamically allocate an NxN integer array laid out in contiguous memory that will function with the bracket notation `array[x][y]`. Specifically the array of N `row_ptrs` should be contiguous in memory and the entire data array of N\*N elements should be laid out in contiguous memory (`data[N*N - 1]` should give you the last element and `data[0]` should give you the first element). `row_ptrs[0]` should be a pointer to the first element in data, `row_ptrs[1]` should be a pointer to the N-th element in `data`. You might not need all the blank lines.



```

_____ allocate_2d_array(int N){
    _____ data = malloc(sizeof(_____) * N * N);
    _____ row_ptrs = malloc(sizeof(_____) * N);
    int i = 0;
    while(i < N){
        _____;
        _____;
        i++; row_ptrs++;
    }
    _____;
    return _____;
}

```

Write code to free all of the memory allocated in `allocate_2d_array`:

```

void deallocate_2d_array(int **row_ptrs){
    _____;
    _____;
}

```

**Q5) MIPS Bit Manipulation (20 pts)**

Complete the MIPS function that **prints AND returns** the number of bits that are different between the upper 16 bits and the lower 16 bits of a 32-bit unsigned number passed in as an argument. Assume there's a print function (defined elsewhere, called `printer`) that prints the one integer argument given.

Please follow calling conventions; you might not need all of the lines below. On a line that is followed by a comment, you must write an instruction that obeys the operation indicated by the comment.

Two examples: `bitsDifferent(0xFFFF0001) → 15`.     `bitsDifferent(0xFFFFFFFF) → 0`.

**bitsDifferent:**

```

    addi _____
    _____
    _____
    _____
    _____
    _____
    li $s0, 0xFFFF
    _____ #logical right shift $a0 by 16, put result in $s1
    _____
    _____
    xor $s1, _____
loop:
    beq _____, $zero, exit
    _____
    _____
    _____
    srl _____, 1
    j loop
exit:
    move _____
    sw _____($sp)
    _____ printer
    _____
    _____
    _____
    _____
    _____ # return control to caller of bitsDifferent

```

**Q6) C ↔ MIPS (15 pts)**

a) Fill in the blanks to translate between MAL MIPS, TAL MIPS, and Machine Code. Wherever you are given space to write both binary and hex for machine code, we will only grade the hex.

Address	MAL MIPS	TAL MIPS
0x102cff00	<b>sll</b> \$v0, \$v0, 5	sll    \$v0, \$v0, 5
	0b _____ == 0x _____	
0x102cff04	_____ \$a0, \$a1, <b>Else</b>	_____ \$a0, \$a1, Else
	0b 000100   00100   00101   _____	
0x102cff08	<b>mul</b> \$t0, \$a0, \$a1	_____
0x102cff0c		_____ \$t0
0x102cff10	<b>j</b> <b>Exit</b>	j        Exit
	0b _____ == 0x _____	
0x102cff14	<b>Else:</b> <some instruction here>	
0x102cff18	<b>Exit:</b> j <b>Far</b>	_____
0x102cff1c		_____
0x102cff20		jr        \$at
	0b 000000   _____   00000   00000   00000   001000	
.....		
.....		
0x20000004	<b>Far:</b> lw    \$v1, 0(\$v0)	lw        \$v1, 0(\$v0)
	0b _____ == 0x _____	